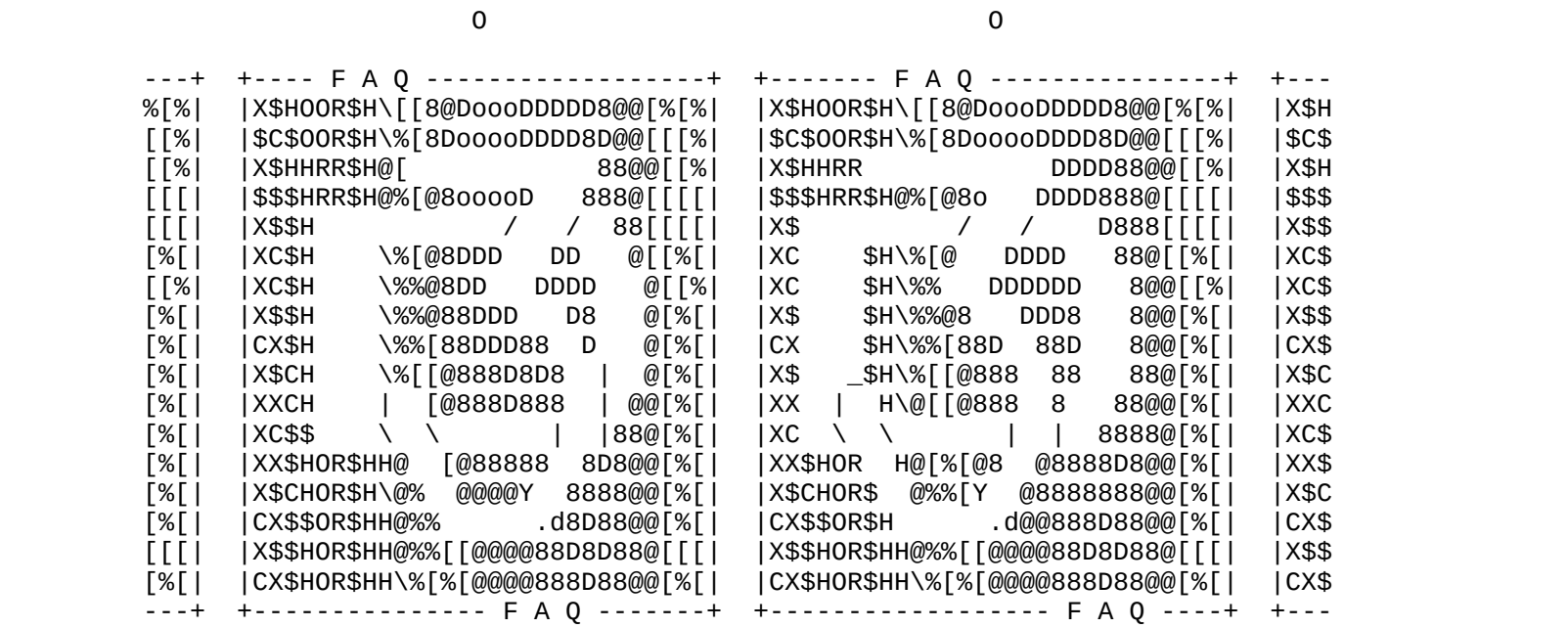
Newsgroups: rec.games.programmer,comp.graphics,rec.answers,comp.answers,news.answers
Subject: FAQ: 3-D Information for the Programmer
Summary: Still in progress, fill-in-the-blanks...
Expires:
From: pat@mail.csh.rit.edu
Reply-To: pat@mail.csh.rit.edu
Sender: pat@mail.csh.rit.edu
Followup-To: rec.games.programmer
Approved: news-answers-request@MIT.Edu
Distribution: world
Organization: Computer Science House @ RIT
Keywords: game-programming 3d-transforms faq
X-Archive-Information: /pub/3dfaq/FAQ.n @ ftp.csh.rit.edu
X-Posting-Information: This article posted automagically, weekly.

Archive-name: 3d-programmer-info
Version: $Id: .header,v 1.8 1994/05/23 15:55:59 pat Exp pat $



Contents:

Last update:
02May96

What's new?

Added list of available 3-D engines

1) General references for 3-d graphics questions.

Well, this FAQ is just getting off the ground. Hopefully it will touch on most of the bases you need to get started for now, and hopefully it will expand at least as fast as you need it too. But... regardless, things you'll want to locate for more help are Matrix Algebra books, Physics books talking about Eulerian motion, and some books on the Graphics Hardware you want to program for. The code examples included in this FAQ will most likely be in C with pseudo-code in comments.

One of the most popular references, (and one of my favorites), is:

Computer Graphics: Principles and Practice

Foley, van Dam, Feiner, and Hughes
Addison Wesley -- Reading, Massachusetts
(c) 1990. ISBN 0-201-12110-7

But, you'll also want to definitely check out the FAQ for comp.graphics. That FAQ touches mainly on 2-D needs, but some 3-D aspects are reviewed there, too.

2) How do I define an object?

There are lots of ways to define objects. One of the most commonly used is the OFF (Object File Format). The OFF toolkit and a library of objects are available via anonymous ftp from gatekeeper.dec.com -- XXX ??? (I can't find it anymore. I found it here once about 2 years ago, but I haven't found it since). The format provides easy methods for extensions and a base set of things you can expect for each object. The toolkit is a bit bulky, but the file format (in ascii) is easy enough to parse by hand.

The OFF.aoff file contains information about the object. The most important one there is the location of the surface specification file (usually object name.geom). This file also contains other attributes and file names relevant to this object.

The OFF surface specification begins with the number of points, the number of polygons and the number of segments.

```
npts nplys nsegs
```

This line is followed by the floating point coordinates for the points that make up the object.

```
x1 y1 z1
x2 y2 z2
x3 y3 z3
   .
x(npts) y(npts) z(npts)
```

Then, it gets a bit more complicated. The following lines begin with a number to indicate the number of vertices in this polygon. That number

is followed by that many numbers, one for each vertex. These are given in an order specified in the .aoff (usually conter-clockwise). So, for example, a triangle and a pentagon which share a side are shown below.

```
        3       1 3 4
        5       2 4 3 6 7
```

Here is some quick and dirty sample code to read in the .geom file:

```
struct polygon {
    int nvert;          /* Number of vertices in this polygon */
    int *verts;         /* Vertices in this polygon */
};

struct object {
    int npts;           /* The number of points */
    int npolys;         /* The number of polygons */
    int nsegs;          /* The number of segments */
    double *point x,*point y,*point z;
                -       -       -
    struct polygon *polys;
};

int
read geom file( char *geom file, struct object *obj )
    -    -                -
{
    FILE *fp;
    int i,j;

    if (!(fp = fopen(geom file,"r")))           /* Open the .geom file */
                        -
        return -1;

                                                /* Get header information */
    fscanf(fp,"%d %d %d",&obj.npts,&obj.npolys,&obj.nsegs);

        /*
        ** Allocate room for the points.
        */
    obj.point x = (double *)malloc(obj.npts*sizeof(double));
              -
    obj.point y = (double *)malloc(obj.npts*sizeof(double));
             -
    obj.point z = (double *)malloc(obj.npts*sizeof(double));
             -

    for (i=0;i triplets, it will
require a fair bit of work on reading them in to turn them into
spherical coordinates. If you're looking to this FAQ for information on
how to define the space your objects will be in, I'd strongly suggest
using rectangular coordinates and some derivative of the OFF-format.

     For starters, let me just throw in that while our universe may be
infinite in all directions, that doesn't make for good programming. We
have to limit ourselves to small enough numbers that we can multiply
```

them together without overflowing them, we can divide them without crashing our systems, and we can add them without accidentally flipping a sign bit.

Now, the fun begins. The simplest form of defining the Universe is to flat out say that the Universe stretches over these coordinates, say in the bounding box of <-65536, -65536, -65536> to <65536, 65536, 65536>. This is often referred to as a Universal Coordinate system or an Absolute Coordinate system. Then, each object in the Universe will be centered about some coordinate in that range. This includes your viewpoint. Several strategies are available for dealing with the edge of the Universe. One can make the Universe wrap around so that an object leaving the cube at < X, Y, 65536> will re-appear in the Universe at < X, Y, -65536>. Or, one can make objects bounce or stop at the edge of the Universe. And, given any approach, one can have the edge of the Universe be transparent or opaque.

In an Absolute Coordinate system, all objects must be shown from the position of your viewpoint. This involves lots of interesting math that we'll get into later. But, in general, an objects position with respect to you is it's absolute position - your absolute position (with all kinds of hell breaking loose if you can see past the edge of the Universe). Then, after this position is calculated, it must be rotated based on your orientation in the Universe.

Another possibility for defining space is a Relative Coordinate system or a View-Centered Coordinate system. In this sort of system, the Viewpoint is always at coordinates <0,0,0> and everything else in the Universe is based relatively to this home position. This causes funky math to come into play when dealing with velocities of objects, but... it does wonders for not having to deal with the 'edge of the Universe'. This is the Schroedinger's cat method of the 'edge of the Universe'.... in the truest sense of out of sight is out of mind. Small provisions have to be made if objects aren't to wrap around. But... a Relative Coordinate system can be used to give the illusion of infinite space on a finite machine. (Yes, even your 486/66DX is finite).

I'll leave spherical coordinates to a later version if people think they'll be of use...

4) How do I define position?

Position in an Absolute Coordinate system is easy. Each object has three coordinates. These are often stored in a data-type called a vector to abstract further the notion that these numbers belong together.

```
typedef struct {
        long x;
        long y;
        long z;

} VECT;
```

Usually, each object in the Universe is defined about its center with each coordinate on its surface being centered at its own <0,0,0>. This helps tremendously in rotating the object, and I would highly recommend this. Then, the object as a whole is given a position in space. When it comes time to draw this object, its points' coordinates get added on to its position.

In a Relative Coordinate system, position is also fairly straight forward. The view-point always has position VECT={ 0, 0, 0 };. Other objects follow the same sort of system that they would in Absolute Coordinate systems.

5) How do I define orientation?

Orientation can be quite tricky. I interchange some of the terms here quite often. In 3-space, orientation must be defined be two-and-a-half angles. "Two and a half?" you say. Well, almost everyone uses three because two just isn't enough, but if you want to be technical, one of those angles only has to range from 0 - 180 degrees (0 - PI radians).

But, taking that for granted now.... you have to pick an orientation for your view. I personally prefer to have the X-axis run from left to right across the center of my screen. I also like to have the Y-axis run from the bottom of my screen; and I also like to have the Z-axis running from me straight into my screen. With some tweaking of plus and minus signs and a bit of re-ordering, all of the math here-in can be modified to reflect any orientation of the coordinate system. Some people prefer to have the Y-axis heading into the screen with the Z-axis going vertically. It's all a matter of how you want to define stuff.

Given that you've agreed with me that Z can go into the screen, what 3-angles do you need? (Here's where I stand the biggest chance of mucking up the terms.) You need roll, pitch, and yaw. (I often mix up roll and yaw and such... so if you can follow along without getting locked into my terminology, future FAQ's will correct it.)

Look at your monitor as you're reading this. Now tilt your head so that your right ear is on your right shoulder. This change in orientation is roll (or yaw... but I call it roll).

Ok, now sit up straight again. Now bring your chin down to meet your chest. (Hmmm... LOOK BACK NOW!!!, whew... glad you heard me.) That motion was pitch.

Ok, now look over your right shoulder keeping your head vertical to see who's behind you. (LOOK BACK AGAIN!!.) Ok... that was yaw (or roll, but I call it yaw).

That's the basics. Now, what do I do with them? Well, here's where a nice book on Matrix Arithmetic will help you out. You have to use these three angles to make a Transformation matrix. [See the section on Matrix Math]. Here is a typical method of doing these transformations: [Note, if you don't have Z going into your screen you'll have to munge these considerably].

```
typedef double matrix[4][4];
double sr,sp,sy,cr,cp,cy;
matrix mr, mp, my;      /* individual transformations */
matrix s;               /* final matrix */

sr = sin( roll );     cr = cos( roll );
sp = sin( pitch );    cp = cos( pitch );

sy = sin( yaw );      cy = cos( yaw );
```

```
                            /* clear all matrixes
                            ** [See the section on Matrix Math]
                            */
        identity( &mr ); identity( &mp ); identity( &my );
                            /* prepare roll matrix */
        mr[0][0] = mr[1][1] = cr;
        mr[1][0] = - (mr[0][1] = sr);

                            /* prepare pitch matrix */
        mp[1][1] = mp[2][2] = cp;
        mp[1][2] = - (mp[2][1] = sp);

                            /* prepare yaw matrix */
        my[0][0] = my[2][2] = cy;
        my[0][2] = - (my[2][0] = sy);

        multiply( &mr, &my, &s );
        multiply( &s, &mp, &s );
```

6) How do I define a velocity?

I've always been annoyed by programs that make it advantageous to run on a slower machine. A naive view of velocity as n-units per time-through-this-loop means that things move faster if you get through the loop faster.

In order to by-pass this problem, you have to keep track of time in some non-hardware-dependent way. Then, you can define your velocity in n-units per known-time-unit. For ease of mental calculations, I always define my velocities in units per second. (In actuality, in programming in DOS, I define my velocity in n-units per 160/182ths of a second because I'd rather divide by 16 than 18.2).

Here is some pseudo-code for an object moving along a line....

```
    int x = 0;                  /* position */
    int vx = 10;                /* velocity */
    int dx;
    int err fact = 0;
           -
    int old time = gettime(); /* in milliseconds for instance */
           -
    int new time;
           -
    int elapsed;

    for (;;) {
        new time = gettime();
           -
        elapsed = new time - old time;
                     -          -
        dx = vx * elapsed + err fact;
                               -
        x += dx/1000;
        err fact = dx - (dx/1000)*1000;
           -
        old time = new time;
           -          -
    }
```

You can ignore all of the err fact stuff if you like, but then
-
velocities that are much lower than your time frequency could end up excessively slow or fast.

This approach can be used for each linear velocity (x, y, and z). But, this approach isn't nearly so good for rotational velocities. I'm still stewing on the best way to handle them.

7) Drawing three-dimensional objects on a two-dimensional screen.

Modified from comp.graphics FAQ:

"There are many ways to do this. Some approaches map the viewing rectangle onto the scene, by shooting rays through each pixel center and assigning color according to the object hit by the ray. Other approaches map the scene onto the viewing rectangle, by drawing each object into the region, keeping track of which object is in front of which.

The mapping mentioned above is also referred to as a 'projection', and the two most popular projections are perspective projection and parallel projection. For example, to do a parallel projection of a scene onto a viewing rectangle, you can just discard the Z coordinate, and 'clip' the objects to the viewing rectangle (discard portions that lie outside the region). To do a perspective projection, dividing each the x and the y by some multiple or the Z-depth is the usual approach.

For details on 3D rendering, the Foley, van Dam, Feiner and Hughes book, reading. Chapter 6 is 'Viewing in 3D', and chapter 15 is 'Visible-Surface Determination'. For more information go to chapter 16 for shading, chapter 19 for clipping, and branch out from there."

8) Vector Math - Dot Product and Cross-Product.

Adding and subtracting vectors is as easy as subtracting their respective parts:

+ =
- =

Scaling vectors is as simple as multiplying each part by a constant:

S * =

The Dot-Product of two vectors is simply the sum of the products of their respective parts:

. = A*D + B*E + C*F

Note that this value is not a vector.

The Cross-Product of two vectors is a bit more complex (it is the determinant of the matrix with the direction vector as the first row, the first vector as the second row, and the second vector as the third

row):

```
         X  =
```

Note that:

```
 X  = -1 * (  X  )
        -and-
( X ) .  = ( X ) .  = 0
```

More later.

9) Matrix Math

The identity matrix is a square matrix (same number of rows as columns) with all elements {i,j} given by:

```
                  { 1.0   if i == j
        m[i][j] = {
                  { 0.0   otherwise
```

Multiplication of matrices:

if X is a matrix that is m rows and n columns (an m-by-n (or mxn) matrix) and Y is a matrix that is n rows and r columns (nxr), then the product X * Y ==> m[i][j] = sum{ a=0, a

Perspective is easy: realize that the viewpoint isn't on the surface of the screen but is some distance back. Then,

[a] convert to a relative coordinate system centered around the viewpoint, oriented so that x and y are like this:

```
                y
                |

             ----+----x

                 |
```

and z is the distance into the screen. Now, divide your coordinates by z.

That's all.

Ok, sure, you don't really have to call your directions x y and z, but that's just changing the description which fits the math. And, sure, you have to decide what to do about things that are "the other direction from the screen" -- but that's really something different.

And, ok, you have to decide how many game units the viewpoint is, behind the screen [this is something you add to the z coordinate before dividing]. And, ok, maybe you want to scale things a bit to make it look nicer. But none of that is really all that hard to deal with.

What's annoying is that you don't want to do division for every pixel that you draw on the screen. That means, for instance, that you might have a line where both ends are off the screen but part of the line [in the middle] are visible. However, that's related to the problem of detecting collisions. Or, better yet, clipping. Yeah, that's it take a look at the section on clipping...

```
Raul D. Miller            n =: p*q                NB. prime p, q, e
                          NB. public e, n, y
                          y =: n&|&(*&x)^:e 1
                          x -: n&|&(*&y)^:d 1     NB. 1 < (d*e) +.&<: (p,q)
```

12) Z-Buffering & the Painters Algorithm & BSP-Trees.

There are several methods available for displaying polygons in a 3-d object so as to assure that what's behind is behind and what's in front is in front. Each, however, has its pros and cons. With each of these methods, it is often useful to implement backface culling. Backface culling is simply not rendering surfaces which aren't facing you.

The Painter's Algorithm is probably the most intuitive for the beginner, one of the easiest to implement, and the hardest to get right. The Painter's Algorithm, quite simply, is Paint the from back to front. Once it has been determined which polygons will be drawn, it is necessary to sort them from furthest away to closest. Then, starting with the furthest polygon and moving to the closest, draw them.

The tricky part of the Painter's Algorithm is determining when one polygon is closer or farther than another. The closest to satisfactory method I've found uses this criteria -- Surface A is further away then surface B if:

1) The furthest point of A is further away than the furthest point of B.
2) B faces you more directly than A.
3) The closest point of A is further than the closest point of B.
4) Surface B is more "important" than surface A.

These are to be taken in order until a decision is reached. But, I've yet to find good enough criteria to work well on most objects that people design. If you keep these criteria in mind when making the objects, it can help. But, if you're using 3rd party objects, I'd suggest a different method.

Z-Buffering is a buzzword that's finally come down to reality. It's been replaced by 'texture-mapping' and 'bump-mapping' and 'realtime raytracing' as the 'what's fancy' buzzword. Z-buffering is costly in terms of memory and processing time, but gives beautiful results. If you've got some freedom with your memory usage and a bit of processor time to spare, and you're ok with filling your own polygons, then Z-buffering may be for you.

To Z-buffer, one keeps a 2d-array of numbers (shorts or ints usually) the same dimensions as the viewport. As each point is prepared for display, the final value is compared to the Z value already in the position in the Z-buffer array. If the new Z is less than the one already in the buffer, the pixel-color is placed in the Screen-Buffer at position and the new Z is copied into position of the Z-Buffer.

The expense of Z-Buffering is having to set each element to the MAX Z VALUE after each frame and keeping track of the Z-value for every
- -
pixel in the Screen Buffer and every pixel in the polygon you're drawing. In most applications, it is unnecessary to keep track of the Z-values of any more than the vertices of a polygon.

And, my favorite of the bunch (but the nastiest to store in an object file, is BSP-trees. BSP-tree stands for Binary Space Partition

tree. They are based the idea that surfaces of most objects don't change positions relative to each other. Objects whose surfaces change relative positions cannot be (to my knowledge) easily used with BSP-trees.

The first step in using BSP-trees is to break up your object into a 'good' object. But, I can't think of a way to describe a 'good' object without just telling you how to make one. So, here's a general algorithm for making a BSP-tree object from ye average object:

1) Pick a surface to call the root node.
2) Compare every other surface in the set of surfaces to the root node.
3) If a surface is In-Front-Of the root node, then put it in the 'Front Heap'.
   If a surface is Behind the root node, then put it in the 'Back Heap'.
   If neither of the above is true, chop up the surface into smaller surfaces that are all either In-Front-Of or Behind the root node. (This is the tricky bit).
4) Repeat these steps with the 'Front Heap' and the 'Back Heap'.

As an interesting note, I can see no advantage at all to trying to keep a relatively balanced tree.

Now comes the interesting bit. Now that the object is a 'good' BSP-tree object, it is time for the drawing algorithm. This is fairly straight- forward.

1) Compare the root node with you.
2) If you are In-Front-Of the root node,
   draw 'Root->Back', then the Root surface, then 'Root->Front'.
   If you are Behind the root node,
   draw 'Root->Front', then the Root surface (unless you're backface culling), then 'Root->Back'.

I fully admit that I didn't believe BSP-trees would work when I first read about them in Foley-van Dam, nor did I believe they would work after seeing them work for me. Now, three months later, I fully believe they work, and I can even say I understand why. I may soon attempt an explanation here, but suffice it for now to say, 'they can be grokked'.

For more info on BSP trees, see the BSP TREE FAQ at http://www.qualia.com/bspfaq/

13) Shading.

Sorry... This is a FAQ in-progress. Will do more later. Please feel free to send me a paragraph for this section.

14) 3-space clipping.

Sorry... This is a FAQ in-progress. Will do more later. Please feel free to send me a paragraph for this section.

15) 3-d scanning.

Three dimensional scanning is a verifiable pain. I've only really heard of three basic techniques. Two of them require the scanned item to be entirely still, and one of them even destroys the object being scanned.

The first technique I ever saw was in 'The Making of T-2'. An

actor was dressed in tight, dark clothing and a mesh of masking tape was placed over his body. This mesh was filmed simultaneously from several different camera positions as the actor walked. The films were then digitally processed to provide a close model of a person walking. The amount of work and expense involved in creating the images, registration of the multiple images, and conversion into 3-space data makes this a big time investment.

The second technique I saw set the item to be scanned on a turntable. A laser sensing system or simple armature-stylus were used to obtain a bump-map along a vertical slice of the item. Then, the item was rotated 5 degrees and the process repeated. The gives a very detailed cylindrical bump-map of an item.

The third technique is by far the most interesting I've seen. But, it is useful mostly only for height-field generation. This method employs a CAT-scan like approach. First, the object is painted solid black (or white). Then, the item is dipped in milk (or chocolate syrup) to a certain depth. A still photo is taken. Then, the item is dipped further into the liquid. Another still is taken. This is done to obtain slice view of the item. The photos are then digitally scanned to locate the boundaries of the items.

16) Publically available source-code.

Well, I've started a collection at ftp.csh.rit.edu::/pub/3dfaq/src. Feel free to upload relevant stuff (that I can't find on archie in under twenty minutes). Some other sites of interest:

3d engines:
http://www.cs.tu-berlin.de/~ki/engines.html

3d images;
ftp://wuarchive.wustl.edu/graphics/ray
ftp://princeton.edu/pub/Graphics

3d Information:
http://www.qualia.com/bspfaq/
http://www.mindspring.com/~cwatkins/algor.html
http://archpropplan.auckland.ac.nz/People/Paul/Paul.html
ftp://x2ftp.oulu.fi/pub/msdos/programming/docs/zed3d023.zip
http://www.csh.rit.edu/~pat/misc/3dFaq.html

3d geometry files:
http://archpropplan.auckland.ac.nz/People/Paul/Paul.html
ftp://ftp.kgc.com/pub/mirror/avalon

ray tracers:
ftp://alfred.ccs.carleton.ca/pub/pov-ray
http://www.mindspring.com/~cwatkins/algor.html

stereograms:
ftp://ftp.comlab.ox.ac.uk/pub/Documents/3d
http://enigma.phys.utk.edu/stereo/index.html
ftp://sugrfx.acs.syr.edu/3d/stereograms

Sorry... This is a FAQ in-progress. Will do more later. Please feel free to send me a paragraph for this section.

17) Books on the topics.

Computer Graphics: Principles and Practice Foley, van Dam, Feiner, and Hughes; Addison Wesley -- Reading, Massachusetts; (c) 1990. ISBN 0-201-12110-7.

I highly reccomend this book for the person seriously interested in understanding Computer Graphics concepts for 2-D image-generation and 3-D representation. As a warning though, if you're struggling to follow vector math and such, you might not spend the $60-$80 bucks on this one yet.

Computer Graphics Handbook: Geometry and Mathematics, Michael E. Mortenson. ISBN 0-8311-1002-3

I've never seen this one, but it comes net-recommended. Anyone care to make a more complete statement?

Programming in 3 Dimensions Christopher D. Watkins and Larry Sharp; Barnes & Noble. ISBN: 1-55851-220-9 $39.95

I've never seen this book. I've got an ad for it in front of me. I would guess it's a very low-density version of some of the 3-D things from Foley. The book boasts sample source code on MS/PC-DOS floppy included. "This one is for all graphics enthusiasts who want a detailed look at 3-D graphics and modeling. Also features discussions of popular ray tracinge methods and computer animation." [sic]

From: MarsSaxMan@aol.com

You mentioned Watkins & Sharp's book "Programming in 3 Dimensions" in your FAQ... I have this book, so I thought I'd give you a capsule review.

I like the book overall. It is cleanly written & they give a pretty good explanation of the mathematics (of course, having spent eons in advanced math has probably bent my perspective on what's clear in mathematics). However it doesn't get too bogged down in theory, which is nice. There are also a bunch of interesting side issues that get covered, such as fractals, color cycling, color tables, etc. Complete source code to a ray tracer, an animation interface to the ray tracer, and some interesting display hacks are included.
Overall a very practical way to get into 3-D graphics; it's pretty easy to see how the code could be modified for this or that purpose. It'll take some work to turn its raytracer into a Doom engine or whatever but it could be done & would still be worth it.

Only thing I didn't really like (though this is probably a natural consequence of the down-to-earth real-world nature of the book) is that it is a bit too DOS PC specific. I hack Macintosh so a lot of the graphics code is unusable and some

of the memory things have to be changed; one chapter on VGA

graphics is wasted on me. But overall I think the book is a good buy. Also it's $39.95, so it's not a bad buy compared to some other 3-D books I've seen (gacckkk!)

Just thought you might like to know, and maybe have a bit more

filler to stash in the FAQ...

-MarsSaxMan Red Planet Software

And, if you liked that... Christopher Watkins sent me a more complete book list (most of which follows). Feel free to send me any personal reviews of any of these books. Like or dislike.

Stereogram Programming Techniques. Christopher D. Watkins and Vincent P. Mallette (Charles River Media, Inc.), 1995 ISBN: 1-886801-00-2 $34.95 Learning Windows(tm) Programming Using Virtual Reality. Christopher D. Watkins and Russ Berube (Academic Press Professional, Inc.), 1994 ISBN: 0-12-737842-1

learning Microsoft Windows(tm) programming by developing a texturing 3-D game engine and application similar to Wolfenstein 3-D(tm) and the popular game DOOM(tm)

Virtual Reality ExCursions with Programs in C. Christopher D. Watkins and Stephen R. Marenka (Academic Press Professional, Inc.), 1994 ISBN: 0-12-737865-0 $39.95

produce polygonal 3-D virtual worlds on your PC, understand human perception issues, near complete list of VR information and technology sources, includes all source code, includes anaglyph 3-D glasses

Photorealism and Ray Tracing in C Christopher D. Watkins and Stephen B. Coy (M&T Books, Inc.), 1992 ISBN: 1-55851-247-0 $44.95

generate photorealistic ray-traced computer graphics, includes all source code for a photorealistic renderer and assorted fractal (and other) database generators, includes many 3-D models

Exploring Photorealism and Ray Tracing Christopher D. Watkins, Vincent Mallette, Stephen R. Marenka and Robert Johnson (M&T Books, Inc.), 1995 ISBN: 1-55851-383-3 $29.95

source code for a beginner's ray tracer, the definitive beginner's book on ray tracing, faster BOB/VIVID executable than in the original, Photorealism and Ray Tracing in C book with even more texture mapping and texturing capabilities, a CD ROMs worth of 3-D models, worlds and tools, various 3-D modeling tools (blobs, polygonal, etc.), learn how to raytrace the beautiful quaternion Julia set fractals (Quaternion -> 4-dimensional object)

Advanced Graphics Programming in C and C++ Roger Stevens and Christopher D. Watkins (M&T Books, Inc.), 1991 ISBN: 1-55851-171-7 $39.95

beginners book to computer graphics, includes a simple ray tracer, polygon renderer, and height-field renderer, and fractals, includes all source code, PASCAL version of this book also available, 1990

18) Other forums.

Sorry... This is a FAQ in-progress. Will do more later. Please feel free to send me a paragraph for this section.

19) Current Contents of Archive ftp.csh.rit.edu::/pub/3dfaq

```
  2 -rw-r--r--    1 pat      member    219 Apr  2 20:40 README
 16 -rw-r--r--    1 pat      member   7787 Mar 28 19:11 DoomTechniques.gz
 10 -rw-r--r--    1 pat      member   4243 May 23 12:34 bsp.gz
 14 -rw-r--r--    1 pat      vr       6266 Apr  2 20:38 imath.gz

 22 -rw-r--r--    1 pat      vr      11262 Apr  2 20:43 imath.tar.gz
```

Pat Fleckenstein
pat@mail.csh.rit.edu

Rob Reay
grendel@mail.csh.rit.edu